Algorithms
Computer Science 140 & Mathematics 168
Spring 2018
Ran Libeskind-Hadas
Handout 1: Good Writing, Induction, and Greed!

1 Introduction

One of the objectives of this course is to practice good writing in the domain of algorithms. This skill is highly valued in both industry and academia. Thus, a substantial fraction of the points on each assignment will be allocated for good writing. Here are the elements that we'll be looking for:

Executive Summary: Always begin with a short overview – or "executive summary" – of the approach that you'll be taking to the problem. This provides the reader with the "big picture" and some context in which to read what follows. Sometimes a single sentence will suffice (e.g., "We will solve this problem with a greedy algorithm, prove that the algorithm is correct, and analyze its worst-case running time."). In other cases, when the problem is more involved, you might need a few sentences.

Begin Paragraphs with an Overview: Break your prose up into relatively short paragraphs just as you would break a long program up into constituent functions. And, just as you would provide short documentation for each function, start each paragraph with a sentence explaining what you're about to do (e.g., "Now we will prove that our algorithm is correct using a proof by induction" or "Now we analyze the running time of our function.").

Use Intuitive Notation: Just as you use good variable names in your programs, use intuitive notation in your algorithm description and proof. While some notation is standard (e.g., G and H are common names for graphs, V and E are common names for the vertex and edge sets in graphs, and u, v, and w are widely-used generic names for vertices in graphs), in many cases you'll need to invent your own notation and variable names. It makes sense to choose notation and names that are easy for the reader to understand and remember. For example, in an

argument about cats and mice, using the variable c for a cat and the the variable m for a mouse is much clearer than using x and y.

Use Pseudo-Code Only When Necessary: In many cases, pseudo-code doesn't actually make your algorithm description more clear. But, if you feel that pseudo-code is helpful, only give it after explaining your code in clear English prose. And, then use very high-level pseudo-code, abstracting away variable declarations and cryptic syntax in favor of adding clear constructs. You'll see examples of this in class.

Read What You Wrote: Please read what you wrote and make an editorial pass before you submit your write-up. Check for spelling and grammatical errors, malformed mathematical expressions, etc. And, almost certainly, you'll find places that you can tweak your write-up to make it clearer and more concise.

When You Know That Something's Wrong: If you know that something is not quite right in your solution, write a caveat at the beginning of your solution that explains what you understand to be less-than-completely correct. This helps us give you good feedback and demonstrates that you understand that there is a problem. *The grading rubric will always give some additional points for thoughtful caveats.* Here's an example: "In the proof below, something isn't quite right in my induction step. The problem arises when ...."

2 Formatting

Please use the LaTeX default margins. (In other words, don't use a margin package such as `fullpage` or `geometry`.) We're counting on wide margins so that we have room to embed our feedback there.

In this document, I used the `amsthm` proof environment. It is one of many LaTeX environments for formatting proofs. If you wish to use this environment and it's not already part of your LaTeX installation, you can search for it online and download it and its documentation. You can also format your proofs in other ways. Just make sure that they are easy to read and include an end-of-proof marker (e.g., a box or a "Q.E.D." at the end – which most proof environments will do for you automatically). This particular environment allows you to say:

```
\usepackage{amsthm}  % Load the proof environment
\begin{proof}
Blah, blah, blah
\end{proof}
```

3 Inductive Proofs

3.1 A First Example

Let's begin with a simple arithmetic result evidently discovered by Gauss when he was in primary school.

Theorem 1. $\forall n \geq 1$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We'll begin with a correct and clearly-written proof of this fact.

Proof. The proof is by induction on n.

Basis: $n = 1$. In this case, the left-hand side is $\sum_{i=1}^{1} i = 1$ and the right-hand side is $\frac{1(1+1)}{2} = 1$ and thus the claim holds for $n = 1$.

Inductive Hypothesis: Assume that the statement is true for $n = k$. In other words, assume that

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

Inductive Step: We must show that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$. By the inductive hypothesis,

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

Therefore, adding $k+1$ to both sides we get

$$\begin{aligned}\left(\sum_{i=1}^{k} i\right) + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\ \sum_{i=1}^{k+1} i &= \frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+2)(k+1)}{2} = \frac{(k+1)(k+2)}{2}\end{aligned}$$

□

Note that in this proof, we introduced the variable k in the inductive hypothesis. This is a stylistic choice and is not at all necessary. Some people feel that this contributes to the clarity of the proof. Others would choose not to introduce this extra variable and would simply state the inductive hypothesis as "Assume that the statement is true for n. In other words, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$." Then, in the inductive step they would say, "We must show that $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$." Either way of writing the proof is considered entirely acceptable.

Next, let's look at a "bad" proof of this same theorem. After the proof, I'll explain what's "bad" about it.

Proof. The proof is by induction on n.

Basis: $n = 1$. In this case, $\sum_{i=1}^{1} i = 1$ and $\frac{1(1+1)}{2} = 1$ and the statement of the theorem holds.

Inductive Hypothesis: Assume that the statement is true for $n = k$. In other words, assume that

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

Inductive Step: We must show that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$.

$$\begin{aligned}
\sum_{i=1}^{k+1} i &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \\
(\sum_{i=1}^{k} i) + (k+1) &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \\
\frac{k(k+1)}{2} + (k+1) &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \quad \text{by the Inductive Hypothesis} \\
\frac{k(k+1)}{2} + \frac{2(k+1)}{2} &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \\
\frac{(k+2)(k+1)}{2} &\stackrel{?}{=} \frac{(k+1)(k+2)}{2}
\end{aligned}$$

Since the left-hand side and the right-hand side of the last expression are equal, the proof is complete. □

What's wrong with this proof? First of all, there is no mathematical symbol $\stackrel{?}{=}$. Using that notation is analogous to writing an ungrammatical sentence. Second, we are starting with what we want to show and deducing something true. This is risky, since it relies on an implicit "if and only if" between each line in the proof above. Although it happens to work out here, imagine that we had done an algebraic step that was not "if and only if". For example, if we had multiplied both sides of one of the lines by 0, we would have deduced in the end that $0 = 0$ which is true. However, we cannot infer from this that the original statement was true.

It's totally fine to use this kind of reasoning in the privacy of your own notes as you are developing the proof. However, when you write up the proof for the world to read, the exposition should be formal and precise. This second proof doesn't cut it.

3.2 More Subtle Induction

Not all inductive proofs will be arithmetic or algebraic in nature. In fact, most proofs in this course will be about data structures or graphs or other mathematical structures. Let's look at another theorem and two proofs of that theorem: this time a bad one first and then a good one.

Recall the notion of binary trees from CS 60. One node is designated as the *root*. A node of degree 0 or 1 is called a *leaf*. A binary tree in which each internal node ("non-leaf node") has exactly two children is called a *regular binary tree*. Recall also that the *depth* of a node is the distance from the root to that node. For example, a regular binary tree is shown in the figure below.

Theorem 2. *In every regular binary tree with n leaves, the number of internal nodes is $n - 1$.*

Proof. The proof is by induction on the number of leaves, n, in the tree.

Basis: When $n = 1$ the tree comprises one leaf and no internal nodes. Thus, the theorem holds in this case.

Inductive Hypothesis: Assume that in every regular binary tree with n leaves the number of internal nodes is $n - 1$.

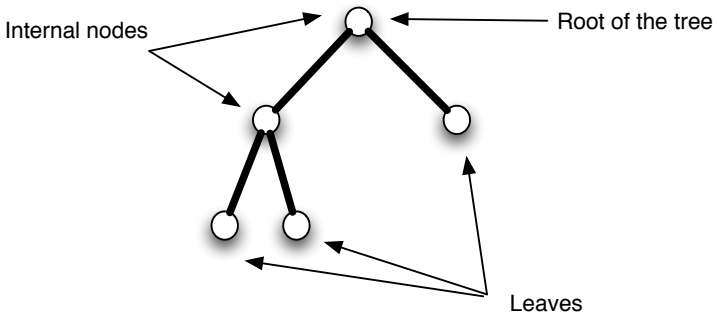

Figure 1: A regular binary tree.

Inductive Step: We must show than in every regular binary tree with $n+1$ leaves there are exactly n internal nodes. Take a tree T with n leaves. By the inductive hypothesis, T has $n-1$ internal nodes. Now, select some leaf of T, call it v. Give v two new children. This results in a new tree T' with $n+1$ leaves since v is no longer a leaf but two new leaves have been added. Moreover, since v is now an internal node, T' has $(n-1)+1 = n$ internal nodes and the proof is complete. □

What's wrong with this proof? The problem here is that in the inductive step we are obligated to show that **every** regular binary tree with $n+1$ leaves has n internal nodes. What we did was to show that **some specific** tree T' with $n+1$ leaves has n internal nodes. One might counter, "well any regular binary tree with $n+1$ leaves can be constructed by starting with some regular binary tree with n leaves and do what we did in the proof – namely turn some leaf into an internal node by giving it two leaves as children." This is indeed true, but we would need to prove that it in order for this inductive proof to be complete. This is almost always difficult and cumbersome. Moreover, there are cases where this kind of augmentation simply does not work; It does not allow us to construct every structure of the next larger size. **This is a common error called the "induction pitfall". Watch out for it!**

So what's the fix? The secret to all happiness is to begin the inductive step by taking some **arbitrary** regular binary tree T' with $n+1$ leaves. We'll then appeal to the inductive hypothesis which applies to *every* regular binary tree with n leaves. Here is how this is typically done:

Proof. The proof is by induction on the number of leaves, n, in the tree.

Basis: When $n = 1$ the tree comprises one leaf and no internal nodes. Thus, the theorem holds.

Inductive Hypothesis: Assume that in every regular binary tree with n leaves the number of internal nodes is $n - 1$.

Inductive Step: We must show that every regular binary tree with $n + 1$ leaves has n internal nodes. Let T' be any arbitrary regular binary tree with $n+1$ leaves. Let u be an internal node of maximum depth. Then the children of u must both be leaves, since otherwise u would have a child which is an internal node of greater depth than u, contradicting the assumption that u was an internal node of maximum depth. Now, construct a new tree T by removing the children of u from T'. Note that u is now a leaf in the tree T and that T is still a regular binary tree since T' was a regular binary tree and no node other than u has changed. Tree T has $(n+1)-2+1 = n$ leaves since two leaves were removed from T' but u has become a leaf. Therefore, by the induction hypothesis, T has $n - 1$ internal nodes. Since T' has one more internal node than T, T' has $(n - 1) + 1 = n$ internal nodes and the proof is complete. □

Let's make a few observations about this proof. First, we began with an *arbitrary* regular binary tree with $n + 1$ leaves since we want to show that the claim holds for every such tree. In contrast, in the previous proof, there was no guarantee that we had considered every possible regular binary tree with $n + 1$ leaves. So, the second proof is better already!

In order to appeal to the inductive hypothesis, we then needed to transform our tree T' into one with n leaves. Although it may seem obvious that we can do this by finding two leaves with a common parent and removing those leaves, this actually warrants proof. We proved this here by identifying an internal node u that is as deep in the tree as possible. This kind of argument is called an *extremal* argument because we are choosing an "extreme" element (in this case "extreme" with respect to depth) to make our proof work. Finally, we reduced T' to a regular binary tree T with n leaves. Since the inductive hypothesis applies to *all* regular binary trees with n leaves, it applies to T in particular, and thus we were able to appeal to the inductive hypothesis to complete the proof.

4 Greedy Algorithms

A greedy algorithm is one that makes a "myopic" simple choice and then recurses on a smaller problem. By "myopic" we mean that the choice is one that can be made based on some property of the individual data elements. Greedy algorithms are generally simple and fast. They are proved correct by mathematical induction. *But, most interesting problems do not succumb to greedy algorithms. So, as tempting as it is to use a greedy algorithm, you must be very careful to prove it correct before you use it.* Let's look at two examples, first the "PIT Registrar Problem" from class and then another problem from our friends at Giggle.

4.1 The PIT Registrar Problem

Problem: The Pasadena Institute of Technology offers an assortment of courses. Each course meets everyday starts and ends at a given time. Our goal is to find a largest set of courses that can be taken concurrently – that is a set of courses whose durations do not overlap.

Solution: We will describe a greedy algorithm for this problem and prove that it is correct.

Short, sweet executive summary.

Our algorithm works as follows: We represent the courses by intervals on the number line where the left endpoint of an interval represents the starting time and the right endpoint represents the ending time. Our goal then is to choose a largest subset of intervals that do not overlap (overlapping just at a start/endpoint is fine). Our greedy algorithm sorts the intervals by ending time. Then we choose the interval with earliest ending time (breaking ties arbitrarily), remove all intervals that intersect with the chosen interval, and recurse on the remaining problem until no more intervals remain.

We're letting the reader know that we're now describing the algorithm.

We now prove the correctness of this algorithm by strong induction on the number of intervals, n.

Alerting the reader to the method of proof.

The basis is when $n = 0$. When there is no interval, our algorithm returns no interval which is clearly correct.

You'll see as you continue reading the proof that it's convenient to have a base case of 0 rather than 1 here.

For the induction hypothesis, assume that our algorithm is correct for any number of intervals between 1 and n.

In the induction step, consider an arbitrary set S of $n + 1$ intervals. Let ℓ be an interval with earliest ending time. We first wish to show that ℓ is in some optimal solution. Assume by way of contradiction that no optimal solution contains ℓ. Then, consider some optimal solution, which we'll denote

Adding one more interval to a set of n would be falling right into the induction pitfall!

by OPT, and let ℓ' be the interval in OPT with earliest ending time. Notice that by definition of ℓ', every interval in OPT other than ℓ' starts after ℓ' ends. Therefore, if we remove ℓ' from OPT and add ℓ, the solution is still valid since ℓ ends at least as early as ℓ'. This new solution has the same size as OPT and thus we have established that there exists an optimal solution that includes the greedy choice ℓ.

At major junctures, remind the reader what we have shown so far.

Let S' denote the set S with ℓ and all intervals that overlap with ℓ removed. Clearly, any optimal solution that includes ℓ must choose the largest subset of intervals from S'. But $0 \leq |S'| \leq n$, and thus by the induction hypothesis, our greedy algorithm will find an optimal number of intervals in this subset. Q.E.D.

Do you see why it was useful to have a base case of $n = 0$ rather than $n = 1$?

5 The Giigle Party Problem!

Problem: You're a summer intern at Giigle. On your first day at work, the CEO comes to your office and tells you about your first task: "I like to throw company parties," she says. "Your job is to write a program to help me figure out who to invite. As you know, Giigle has a hierarchical structure. You can think of it as a tree. The CEO, that's me, is at the root of the tree. Below the root are supervisors, below them are managers, below them are team leaders, etc., etc., until you get down to the leaves - the summer interns. The tree is not necessarily binary; some non-leaf nodes may have one "child", others two, and others even more. Anyhow, to make the party fun, I thought it best that we don't invite an employee along with their immediate boss (their parent in the tree). My objective is to invite as many employees as possible without inviting an employee and their boss."

Describe a **greedy algorithm** for this problem in clear English. Your algorithm should take a tree as input and return the largest set of employees in the tree that can be invited to the party subject to the constraint that if an employee is invited, their boss is not invited. You should assume that the tree is stored as a collection of nodes where each node has a reference to its parent, a count of the number of its children, and a list of references to its children. You may assume that the size of the tree is also given. Prove the correctness of your algorithm and give its worst-case running time.

Solution: We describe a greedy algorithm, prove that it is correct, and analyze it's running time.

Our greedy is based on the observation, which we will prove below, that it's always good to invite the leaves of the tree to the party. Our greedy algorithm, therefore, begins by choosing some arbitrary leaf, v, and inviting it. Next, the parent of v, denoted $p(v)$, must be removed from consideration. Thus, we remove $p(v)$ and all its incident edges. This results in fragmenting the original tree into one or more smaller trees (including some that might be single vertices - e.g., any leaf siblings of v). Each of these smaller trees is now solved independently using the greedy algorithm.

We prove the correctness of this greedy algorithm using strong induction on the number of vertices n in the tree.

The basis is when $n = 1$. There is only one optimal solution (take the vertex!) and our greedy algorithm finds it.

The induction hypothesis is that the greedy algorithm is optimal for any tree with n or fewer vertices.

Now, in the induction step, we consider an arbitrary tree with $n + 1$ vertices. The greedy algorithm chooses a leaf v. To begin, we will show that the leaf v that we chose is part of some optimal solution. To show this, consider some optimal solution OPT. Note that OPT is a set of vertices in the tree that can be invited to the party and, by assumption, there is no larger set than OPT that can be invited to the party. If $v \in OPT$, we have shown that v is in some optimal solution. If $v \notin OPT$ then consider $p(v)$, the parent of v. If $p(v) \notin OPT$ then OPT is not optimal since we could invite v after all and increase the size of the solution. So, $p(v) \in OPT$. Then, removing $p(v)$ and adding v to OPT is still a valid solution and it's a solution of the same size as OPT and is thus an optimal solution. So, we now know that there exists an optimal solution that includes v.

The inclusion of v requires that we remove $p(v)$ and all edges incident on $p(v)$. This results in some number of smaller trees. Each of these smaller trees is independent of the other trees since they have no edges in common (we've removed those edges when we removed $p(v)$). Therefore, an optimal solution should choose the largest number of employees from each of those smaller trees. Our greedy algorithm will recurse on each of those trees and we know, by the induction hypothesis, that the greedy algorithm will find the optimal solution in each of those trees. Thus, the greedy algorithm finds an optimal solution to the original problem. Q.E.D.

The executive summary in this case is basically repeating what we were told to do in this problem. In more complex problems, the executive summary is likely to provide some additional insight into what we're about to do.

We're giving some motivation for our approach before getting started.

Note that we could invite all of the leaves simultaneously, but our proof of correctness will be somewhat simplified by inviting one leaf at a time.

Alerting the reader to the method of proof.

Notice that we're avoiding the infamous induction pitfall! Rather than adding a new vertex to a previously considered tree, we're choosing an arbitrary tree with $n + 1$ vertices.

Again, alerting the reader to what's coming next.

In other words, the choice of v is **safe**!

Indicate that the proof is done.